Engineering Challenge

Build a system, in the language of your choice, for (1) processing events representing changes to the state of an object and (2) creating a system for efficiently representing that object's state over time.

Your system will need to track change to the following `UserProfile` object (note that here we list the objects keys and value types):

```
{
      “id”: String                      # UUID (RFC 4122)
      “username”: String                # Users username
      “bio”: String                     # Users bio
      “createdAt”: String               # ISO 8601 Timestamp
      “updatedAt”: String               # ISO 8601 Timestamp
      “lastLogin”: String               # ISO 8601 Timestamp
      “followerCount”: Int              # Number of people following user
      “followingCount”: Int             # Number of people user is following
      “remainingInvites”: Int           # Number of invites user can send out
      “bookmarks”: String List          # UUID (RFC 4122) list of bookmarked items
      “phoneNumber”: String             # E.164 formatted phone number
}
```

These records are created and all of the fields except `id` and `createdAt` can be updated during the life of the `UserProfile`. For this problem you do not need to worry about the `UserProfile` object being deleted, and can assume we want to keep related information indefinitely (in the case of a deletion you can assume that events will simply stop being generated for this user).

When a new `UserProfile` is created, an initial event is fired which contains the type `CREATE` and any fields which were available when the `UserProfile` was created (this event will always include `id`, `type` and `createdAt` fields).

```
{
      “id”: “5017665a-80e9-4b1f-ae2e-ee2609f2fb5e”     # UUID (RFC 4122)
      “type”: “CREATE”,
      “fields”: {
            “createdAt”: String       # ISO 8601 Timestamp
            ...                       # Other fields from the above list
      }
}
```

When a user updates their `UserProfile`, or a function of the system causes their `UserProfile` to be updated, then it instead creates an event of the type `UPDATE`. The event will also contain the name of the fields that were updated as well as specific value information based on the type. `String` type fields will have an `old` and `new` field, which contain the value before and after the update (respectively). `Int` type fields will have a `type` field which can be `INC` for increment or `DEC` for decrement and a `value` field which is how much the integer is being incremented or decremented by. `String List` type fields will have an `added` field, which is a list of items that are being added to the aray, and a `removed` field, which is a list of items that are being removed from the array. You can assume added values are appended to the end and that removed values always exist in the list its being removed from..

```
{
	“id”: “5017665a-80e9-4b1f-ae2e-ee2609f2fb5e”     # UUID (RFC 4122)
	“type”: “UPDATE”,
	“fields”: {
		“bio”: {
			“old”: “my old bio”,
			“new”: “my new bio”
		},
		# All other String updates follow bio’s format above
		# If there was no previous value only the “new” field will exist
		“followerCount”: {
			“type”: “INC”      # INC for increment, DEC for decrement
			“value”: 3         # The amount incremented or decremented
		},
		# All other Int type updates follow followerCount’s format above
		“bookmarks”: {
			“added”: [“11eedbdb-8ea1-4823-afbf-cbd2228d5a73”]
			“removed”: []      # Nothing removed in this case
		}
		# There are no other lists in the UserProfile except bookmarks
		# “added” is an array of items added to the bookmarks list
		# “removed” is an array of items removed from the bookmarks list
	}
}
```

You can assume that the above events are processed from a queue, so they will be received one at a time and in the correct order. To simulate this the events will be provided to you through an `events.json` file with the following format (we have provided an example file here):

```
{
	“Events”: [
		{
			“id”:
			“type”:
			“fields”: {
				# Fields for the first event
			}
		},
		{
			“id”:
			“type”:
			“fields”: {
				# Fields for the second event
			}
```

```
            },
            # All the other events, similar to above
      ]
}
```

Your program should read this file and create a Point in Time (PIT) object viewer. Your PIT viewer should create a structure that allows for random access querying of any object at any point in the application of the events to the object. You can assume that accessing the object at PIT `0` is the object as it was created by the `CREATE` event and accessing the object at PIT `N-1` (where `N` is the number of events that were applied to that object) is the object after all events have been applied to it.

While the example file we have provided is small, the system should be designed to handle these events at scale. You should consider both the size necessary for storing your objects over time as well as the performance of the queries. You should discuss how these considerations affect the design of your system in the README.

Using the example `event.json` we would get the following outputs:

`UserProfile` object `1ebe7809-b708-46b0-b0b6-7d8f608ff08e` at PIT `0`

```
{
      "id": "1ebe7809-b708-46b0-b0b6-7d8f608ff08e",
      "createdAt": "2020-10-28T03:20:02Z",
      "updatedAt": "2020-10-28T03:20:02Z",
      "username": "the-first-user",
      "phoneNumber": "+19095550101",
      "bookmarks": [
            "da1028ad-43da-42ec-b549-dbb17963b54f",
            "6ccb2eae-cb7c-445d-96e4-0af13f04fed2",
            "28787d49-dd06-4118-8cca-444eb0cee3cc"
      ],
      "bio": "User #1!"
}
```

`UserProfile` object `1ebe7809-b708-46b0-b0b6-7d8f608ff08e` at PIT `2`

```
{
      "id": "1ebe7809-b708-46b0-b0b6-7d8f608ff08e",
      "createdAt": "2020-10-28T03:20:02Z",
      "updatedAt": "2020-10-28T03:24:02Z",
      "lastLogin": "2020-10-28T03:24:02Z",
      "username": "the-first-user",
      "phoneNumber": "+19095550101",
      "bookmarks": [
            "6ccb2eae-cb7c-445d-96e4-0af13f04fed2",
      ],
      "bio": "No really, I'm user #1"
      "remainingInvites": 5
}
```

`UserProfile` object `770375c4-8273-4c6f-9c68-7e55f7174c21` at PIT `1`

```
{
	"id": "770375c4-8273-4c6f-9c68-7e55f7174c21",
	"createdAt": "2020-10-28T03:22:02Z",
	"updatedAt": "2020-10-28T03:23:02Z",
	“lastLogin”: “2020-10-28T03:23:02Z”
	"username": "the-second-user",
	"phoneNumber": "+19095550102",
	"followerCount": 20,
	"followingCount": 15,
	"bio": "User #2!?"
}
```

Deliverables

- Production quality code
    - Follows standard syntax and patterns of the language
    - Has test coverage to ensure correctness and quality
    - Well commented and/or documented as appropriate
- A README explaining the following:
    - How to build and execute your code (as appropriate)
    - The general strategy/approach you took towards testing
    - The rationale behind the choices you made in the design of your PIT viewer
    - Any assumptions you made about the system and your reasoning behind it
    - Anything else you would like us to know